UrsaX: Integrating Block I/O and Message Transfer for Ultra-Fast Block Storage on Supercomputers

Shun Gai, Yiming Zhang, Xuchao Xie, Haowen Chen, Xi Zhao, Yong Dong, and Zhenlong Song

***Abstract*—It is increasingly important for the next-generation exascale supercomputers to extend its applications beyond traditional HPC (high-performance computing) scenarios, so as to achieve high social and economic benefit. Similar to AWS (Amazon Web Services) and Alibaba Cloud, cloud-style virtual HPC service is a promising application scenario on supercomputers, for which remote block storage is the key to provide tenants with supercomputers' extremely-high storage performance. Unfortunately, the state-of-the-art block storage software systems (such as URSA and Ceph) cannot adapt to the advanced hardware features of supercomputers.**

This paper presents UrsaX, an efficient block storage service for our next-generation Tianhe exascale supercomputer that is equipped with the high-performance GLEX network and NVMe (Non-Volatile Memory Express) SSDs. UrsaX's virtual disks, which can be mounted like normal physical ones, enable not only traditional HPC applications but also supercomputer-oblivious POSIX applications to enjoy the high performance of supercomputers. At the core of UrsaX is with a novel design of the efficient integration of on-disk block I/O and in-network message transfer on supercomputers. UrsaX utilizes the NVMe Fabrics kernel module to expand the NVMe standard on the supercomputer network, and separates metadata I/O and data I/O of blocks respectively being handled over the MP (Mini Packet) and RDMA (Remote Direct Memory Access) protocols. We thoroughly explore the design space for remote block storage on supercomputers including parallelism, scalability, fault tolerance, and consistency. We conduct extensive evaluation on a subset of our exascale supercomputer consisting of 44 storage machines (each with four NVMe SSDs). The result shows that UrsaX achieves local-storage-level I/O latency (tens of microseconds) while being able to linearly increase the aggregate performance (IOPS and throughput) as the system scale increases, an order of magnitude higher than the state-of-the-art block storage systems.

***Index Terms*—Block storage, supercomputers, block I/O, message transfer.**

I. INTRODUCTION

WHILE it has been a long time for high-performance supercomputing to be a national strategy for many developed countries, currently the increasingly high monetary cost requires supercomputers to provide more social and economic benefit to the public. With the development of supercomputing technology, it is more and more important for the next-generation exascale supercomputers to extend the application scenarios in addition to the traditional HPC (high-performance computing) applications.

Shun Gai, Xuchao Xie, Haowen Chen, Xi Zhao, Yong Dong, and Zhenlong Song are with National University of Defense Technoogy (NUDT), Yiming Zhang is with Xiamen University and NUDT. Shun Gai and Yiming Zhang are co-primary authors.

Manuscript received July 16, 2022.

In recent years, we see a trend of fusion between supercomputing and cloud computing [7], [31]. By integrating existing virtualization techniques for the cloud such as VMs (virtual machines) and containers, supercomputers can realize efficient and reliable resource sharing, scheduling, and isolation. For example, Tianhe-2 supercomputer [31] has adopted Slurm [29] to efficiently schedule jobs from thousands of tenants and substantially improve its resource utilization, and Cray Shasta exascale supercomputer [7] has adopted Kubernetes containers [10] so as to fully support conventional cloud-native services.

The cloud services running on the Tianhe and Cray supercomputers demonstrate that cloud-style virtual HPC service is a promising application scenario for supercomputers, and currently many POSIX applications that have traditionally reside in the cloud are being migrated to supercomputers. Various applications require different types of resources. For example, computing-intensive applications require high computing power, and I/O-intensive applications require high bandwidth and throughput. Similar to the cloud, supercomputers can support these VM-/container-based applications by providing them with high-performance storage through virtual disks (*i.e.*, volumes) which are backed by remote block storage [26] that has been widely applied in the cloud, such as Amazon EBS (Elastic Block Store) [2], Microsoft Azure Storage [3], Tencent QCloud [19], OpenStack Cinder [14], and Alibaba Cloud Storage [8].

Virtual block storage service is the key to deliver supercomputers' extremely-high performance to the tenants. Although in the past the performance of cloud networks had been the bottleneck for block storage performance, modern supercomputer networks provide high bandwidth (several hundred Gbps) and low latency (tens of microseconds), making it possible for remote volumes to even outperform local disks. On the downside, however, supercomputing usually has distinct requirements on block storage compared to traditional cloud computing. First, HPC jobs require much higher IOPS and lower latency than traditional cloud tasks. For instance, trillion-scale graph processing needs to read graph data from storage extremely fast, otherwise data I/O would become the bottleneck. Second, for supercomputing most CPU cycles are expected to be used for "real computation" rather than for I/O, while I/O-intensive HPC jobs tend to cause high CPU usage.

Unfortunately, the state-of-the-art cloud block storage systems (such as URSA [26], Ceph [40], and Sheepdog [13]) cannot meet the requirements when being migrated to supercomputers. This is mainly because they are agnostic to the underlying supercomputing architecture, thus being unable to leverage the advanced HPC hardware features such as

RDMA (Remote Direct Memory Access) and NVMe-oF (Non-Volatile Memory Express over Fabrics) for high-performance I/O. Note that directly modifying existing storage systems (like Ceph) to support RDMA and NVMe-oF only yields marginal improvement [36], as simple integration is suboptimal to both the storage stack and the network stack (Section V-A).

To satisfy the stringent requirement of supercomputer tenants on the performance, reliability, scalability, and availability of remote volumes, this paper presents UrsaX, an efficient block storage service for our next generation Tianhe exascale supercomputer that is equipped with the high-performance *GLobal EXpress* (abbreviated as GLEX) network and NVMe (Non-Volatile Memory Express) SSDs. UrsaX's virtual disks, which can be mounted like normal physical ones, enable not only traditional HPC applications but also supercomputer-oblivious POSIX applications to enjoy the high performance of supercomputers. At the core of UrsaX is the efficient integration of on-disk block I/O and in-network message transfer by leveraging the advanced storage/network hardware features of supercomputers. UrsaX utilizes the NVMe-Fabrics kernel module to realize the extension of the NVMe standard on the supercomputer network, which adopts NVMe-oF with the model-based messaging mechanism to send I/O requests and receive responses over the network. UrsaX separately handles block I/O's metadata and data respectively over MP (Mini Packet) and RDMA protocols. Further, UrsaX implements an adaptive interrupt aggregation mechanism in firmware to reduce CPU usage for I/O intensive workloads.

We have implemented UrsaX for our exascale supercomputer, by thoroughly exploring the design space for high-performance block storage including parallelism, fault tolerance, and consistency. UrsaX adopts the master-target-host architecture where one or more masters maintain the metadata, a group of target storage servers store data blocks, and multiple host servers (acting as the surrogate for client tenants) communicate with the masters and target servers. UrsaX adopts replication at the target storage servers for data durability. We conduct both micro benchmarks and application experiments on a subset of our exascale supercomputer consisting of 44 storage machines (each with four NVMe SSDs). The result shows that UrsaX achieves local-disk-level I/O latency (tens of microseconds) and can horizontally increase the aggregate performance (IOPS and throughput) as the system scale increases, being an order of magnitude higher than existing cloud storage systems.

The contribution of this paper is summarized as follows.

- We discuss the necessity of realizing cloud-style block storage service on supercomputers and identify the characteristics of the workload of block storage service.
- We leverage the advanced storage and network hardware features of supercomputers to efficiently integrate on-disk block I/O and in-network message transfer, so as to achieve high I/O performance on supercomputers.
- We realize interrupt aggregation so as to reduce CPU usage for I/O intensive workloads without affecting I/O latency of lightweight workloads. Unlike existing solutions [33], [34], [41] which break the NVMe specification thus being impractical for real usage, our proposal is completely compatible with the NVMe specification.
- We collaboratively design the management, replication, parallelization, and fault tolerance for block storage on supercomputers, which achieve high reliability, scalability, and availability without compromising performance.

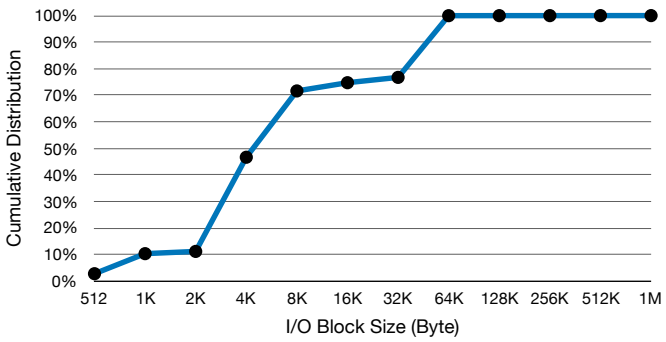


Fig. 1. CDF of I/O block sizes.

II. Background and Related Work

A. Block Storage

Distributed block storage systems [23], [25], [27], [39] have been widely deployed in the cloud to provide virtual disks to the tenants, such as Amazon EBS [2] and Microsoft Blizzard [28] which are backed by high-performance solid-state drives (SSDs). They provide a block interface to remote clients using protocols like iSCSI. For example, Petal [25] uses redundant network storage to provide volume service. Salus [38] leverages HDFS [6] and HBase [22] to provide volume service with ordered-commit semantics using a two-phase commit protocol. pNFS (parallel NFS) [23] exports a block/object/file interface to local cloud storage. Blizzard [28] is built on FDS [30] and exposes disk parallelism to volumes with crash consistency guarantees.

To reduce the monetary cost of cloud block storage, recently URSA [26] adopts an SSD-HDD-hybrid storage structure which stores primary data on SSDs and places backup data on hard-disk drives (HDDs). The performance gap between SSDs and HDDs is bridged by using small journals on SSDs with a log-structured merge-tree (LSMT) [32] index. URSA also support the SSD-only mode that stores both primary and backup data on SSDs.

To understand the characteristics of the workload of block storage service, we collect a one-month trace of our block storage cluster of the Tianhe exascale supercomputer, where clients mount virtual disks to run their HPC/POSIX applications. We first calculate the cumulative distribution of all I/O block sizes. The result is shown in Fig. 1, demonstrating that random small I/O is dominant. Note that (although being rare) large sequential I/O does exist, *e.g.*, for checkpointing of HPC applications and for file backup of POSIX applications. We also measure the aggregate I/O load overtime. The result (not depicted due to lack of space) shows that the workloads (demonstrating the aggregate I/O of all applications) have a

repeating pattern every day, and most of the time the aggregate loads are significantly lower than the per-day peaks.

The workload characteristics require the block storage service to (i) efficiently support random I/O which is dominant, and (ii) flexibly map (*i.e.*, place) data blocks onto storage devices as the changing load needs adaptive service expansion/shrinkage at low cost. Unfortunately, the state-of-the-art cloud block storage systems (including URSA, Ceph, Sheepdog, EBS, Blizzard, etc.) are unable to leverage the advanced HPC hardware features such as RDMA and/or NVMe-oF. For instance, Disaggregated Flash [24] uses the iSCSI protocol to organize remote flash SSDs into a flash storage tier allowing clients to access its key-value service via TCP/IP, but it cannot take advantage of supercomputers high-performance RDMA networks. Although some studies have recently migrated Ceph onto RDMA networks [36], they usually use hashing to map data blocks to storage devices, which prevents flexible mapping management. Specifically, Ceph block storage (Ceph RBD) [4] is built upon the RADOS object store [5] that adopts hash-based CRUSH [40] for data block placement, which suffers from heavy data migration and severe performance degradation when expanding/shrinking the system [37]. In addition, CRUSH prevents *topology-aware* mapping, *i.e.*, using the storage devices of nearby nodes of the client as the virtual disk backend, and thus results in suboptimal I/O performance on supercomputers that have extremely-large networks with high diameters.

B. NVMe over RDMA

NVMe SSDs achieve very high local I/O performance (several GB/sec bandwidth and tens of microseconds latency). To enjoy the high I/O performance of NVMe across the network, researchers extend the original NVMe over PCIe protocol to propose NVMe over Fabrics (NVMe-oF), which supports mapping NVMe to multiple Fabrics transmission options such as FC, InfiniBand, RoCE, iWARP and TCP. NVMe transmission is an abstract protocol layer for reliable NVMe command and data transmission. NVMe-oF replaces PCIe to extend the communication distance between the NVMe host and the NVMe storage subsystem.

Among these Fabrics option protocols, InfiniBand, RoCE and iWARP are more attractive because they support RDMA (Remote Direct Memory Access), an efficient remote memory access technology that allows a computing node to directly access the memory of other nodes without time-consuming processing by CPUs. RDMA quickly moves data from the memory of a node to the memory of a remote node without involving the operating systems, which provides NVMe-oF with local-disk-like I/O performance (low latency and high throughput).

RDMA is designed for offloading data transfer to RDMA NICs (RNICs), by specific operations like RDMA GET and RDMA PUT. A GET/PUT option supports a maximum of 128 MB data transfer. Specifically, at the receiver side, when a GET descriptor is submitted, RNIC will pack it into a packet, which will be transferred through the network and be written to the memory of the sender by its RNIC (bypassing CPU). Similarly, at the sender side, when a PUT descriptor is submitted, RNIC will read the described data from memory, pack the data into packets, and send the packets through the network, which will be written to the memory of the receiver by its RNIC. Event or interrupt can be used to indicate the completion of data transfer.

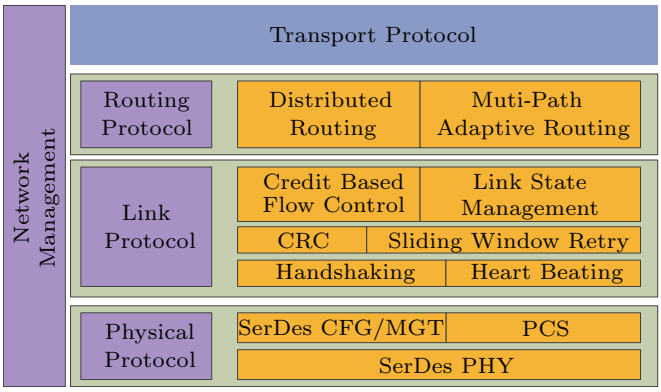


Fig. 2. GLEX network stack.

C. The GLEX Network

Supercomputers usually rely on InfiniBand-like high-speed networks for fast interconnection. For example, our supercomputer adopts the *GLobal EXpress* (abbreviated as GLEX) interconnection network, which realizes user-level communication by leveraging the memory management unit (MMU) in CPUs and RNIC bypassing OS in the critical communication path. GLEX adopts a layered network stack (Fig. 2). Similar to the TCP/IP network stack, The GLEX network stack consists of multiple levels including the physical level, link level, and routing level.

III. DESIGN

A. Overview

As the fusion between supercomputing and cloud computing is emerging, we design the UrsaX block storage service for supercomputers. UrsaX accommodates both traditional HPC applications and supercomputer-oblivious workloads that previously run in the cloud. Fig. 3 depicts the high-level overview of UrsaX, which is composed of three key parts, namely, GLEX network, host servers, and target servers. The host servers act as a portal or surrogate to which the entire I/O requests from applications are sent, while the true NVMe storage is realized by the target servers. The GLEX network, which is designed for high bandwidth and low latency requirements and has been adopted by our supercomputer, is responsible for the communication between host servers and target servers. Both host servers and target servers have a specific module (which works in the transport layer of the network) for data transmission, namely, the `nvme-kglex` module in host servers and the `nvmet-kglex` module in target servers. For example, the heartbeat signal is used for host servers to make sure target servers is alive, and the discovery requests is for host servers to mount a virtual disk.

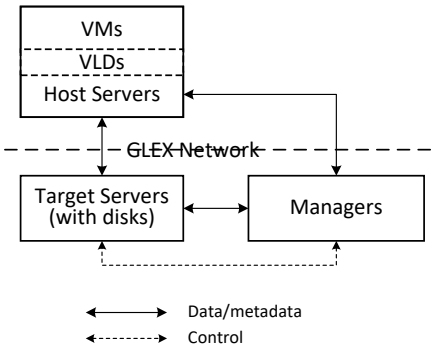


Fig. 3. Host servers and target servers in UrsaX.

Since a volume is mostly used by only one client tenant, we can optimize for this simple use case which is dominant for supercomputer-based block storage. Consequently, durability and strong consistency guarantees can be achieved without lowering the I/O performance: we adopt n-way replication (by default $n = 3$) for durability; and if a write to a data block is committed, we guarantee that afterwards any following read for that block will see the committed data, which will be introduced in detail in Section IV.

UrsaX decouples backend real storage from frontend virtual storage (like replication, thin provisioning, striping, etc.) by mapping the physical disks of the target servers onto the host servers. The mapping between the frontend and backend storage is maintained by the managers (introduced in detail in Section IV-A). Target servers are responsible for real block storage, and host servers act as a portal for processing block I/O requests from tenants and can also run tenants' client applications.

Host servers can create volumes (*i.e.*, VLDs) with requested capacity from tenants. Target servers provide physical SSDs to back the VLDs. A VLD can be mounted as a block device with the name like /dev/nvmenXpY which can be discovered by the tenants. The size of a VLD is specified when a client tenant creates the VLD (via a host server), but the real storage space on physical disks of target servers is dynamically allocated. When opening a VLD, the host server will ask the managers to get the meta information of the VLD.

For the first time a logical chunk is written, the host server will ask the storage service manager to allocate n physical chunks from n least-loaded physical disks each in one failure domain. By default UrsaX set $n = 3$, *i.e.*, three-way replication where one replica is primary and the other two are backup, which will be introduced in the next subsection.

Note that although currently the design of UrsaX is tightly coupled with the underlying GLEX network, it will be relatively straightforward to migrate UrsaX to other supercomputers which are built on top of InfiniBand networks, by using the general RDMA protocol to replace the special MP protocol of GLEX (at the price of slight performance loss for mini packets).

In the rest of this section, we will present the basic design of UrsaX, mainly including (i) how we separate host and target servers and use NVMe-oF to integrate them that collaboratively provide virtual local disks or VLDs (Section III-B), (ii) how we efficsiently support read/write requests by leveraging the GLEX network's features (Section III-C), and (iii) how we aggregate interrupts to reduce interrupts without affecting I/O performance (Section III-D).

B. Host Servers and Target Servers

In UrsaX, host servers serve as the storage service portal. All the I/O requests from tenant applications are sent to host servers which provide the mounted VLDs. These VLDs are backed by the corresponding target servers' physical disks. Host servers receive requests not only from applications but also from target servers, and accordingly process these requests such as responses of discover requests, connect requests, disconnect requests, NVMe command capsules and I/O requests. Host servers can create new VLD devices, such as /dev/nvmenX, by NVMe connect commands. Meanwhile, the corresponding target servers will create the devices for every create command from host servers. The record of a VLD contains a unique ID (*i.e.*, NVMe Qualified Name or shortly NQN) and the IP address of the corresponding target server. By the IP address and NQN, we can locate the corresponding device mounted on a specific target server uniquely.

As shown in Fig. 4 and Fig. 6, when an I/O requests is sent to the host server, it will be processed by nvme-core into one or more blk-mq (block mq) requests, each of which will be packaged into an NVMe command. The nvme-kglex module of the host servers, collaborated with the nvme-core, blk-mq and GLEX driver, handles these commands to communicate with the target servers. As depicted in Fig. 8, these commands are inserted into a lockless list, and then sent to the target servers by a thread (I/O WORK). The responses from target servers are stored in an in-memory ring lockless queue temporarily, and are fetched by a thread (MSG WORK) to complete the write request. In practice, the numbers of NVMe SSDs and dedicated CPU cores are configured to match the network speed so that the commands in the ring lockless queue does not need to wait. The two threads (MSG WORK and I/O WORK) run on the same core.

Target servers install physical disks to back the VLDs. Target servers split the entire disk space into fixed-size chunks (by default 64MB) that will be dynamically allocated to VLDs on demand, according to the requests from host servers. Target servers only accept requests from host servers, mainly including I/O requests, create and destroy requests, etc. As shown in Fig. 5 and Fig. 7, once a target server receives a request, which will be stored by a ring lockless queue temporarily, it is the duty of the thread MSG WORK to fetch request from ring-buffer and resolve it. I/O requests will be inserted into blk-mq waiting for processing; discover requests will be transmitted to module nvme-discovery; and other requests will be transmitted to nvme-core. The MSG WORK thread constructs the response packet according to the processing result, and inserts it into the send waiting queue which is also lockless. The thread calling I/O WORK is responsible for fetching response from lockless list and sending it to the corresponding host server. On target servers, the two threads (MSG WORK and I/O WORK) work on the same core.

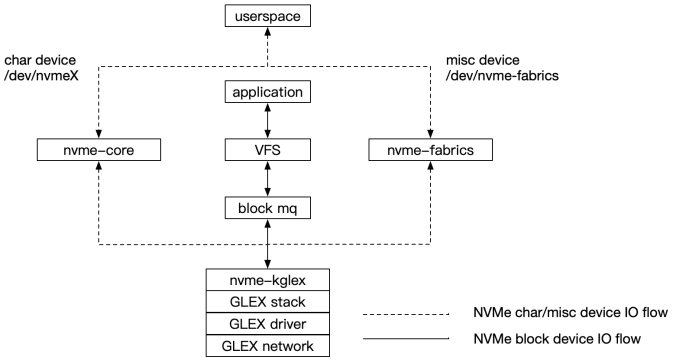


Fig. 4. Host server architecture.

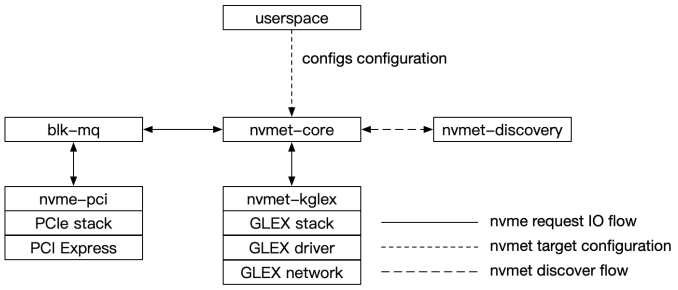


Fig. 5. Target server architecture.

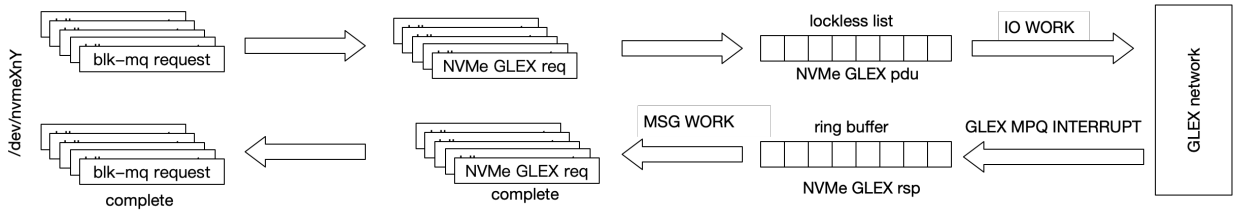


Fig. 6. Handling I/O requests in the nvme-kglex module on Host Server.

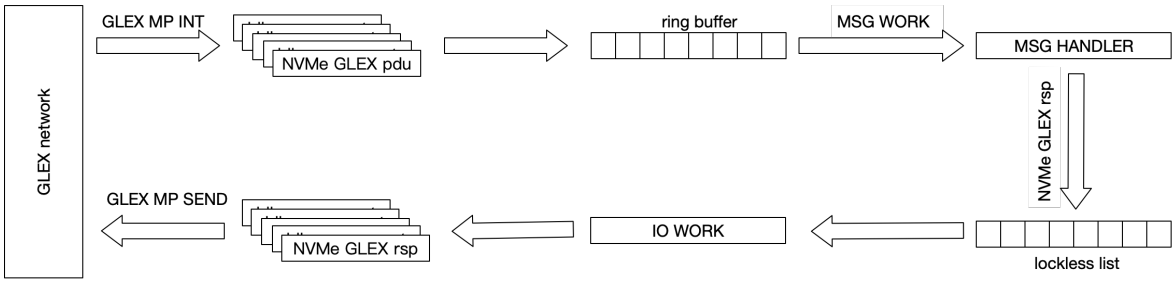


Fig. 7. Handling I/O requests in the nvmet-kglex module on Target Server.

C. Reads and Writes over Network

UrsaX mounts VLDs on host servers and constructs VFS on virtual devices. Target servers' disks serve as the backend for the remote VLDs uniquely identified by NQNs. Further, data transmission (including requests, commands, and data) between host and target servers is realized through the GLEX network.

At the user level, GLEX provides users with the MP and RDMA transfer operations, which are designed to be non-blocking so as to support the overlap of communication and computation in the applications. As introduced in Section II-B, the RDMA operations can directly transfer large bulks of user-space data bypassing the OS. Different from RDMA, MP is used for mini-packet transfer. When the data size is smaller than a threshold (120 bytes configured in UrsaX), the data can be packed into the descriptor instead of into the memory, which allows UrsaX to save memory access times and realize low-latency communication for small packets. After an MP descriptor is submitted, RNIC can obtain data from the descriptor and directly transfer the packet through the GLEX network. When the MP packet arrives at the destination, it will be written into the corresponding in-memory MPQ (MP packet queue). The receiver detects MP packet arrivals by polling or interrupt. UrsaX adopts the MP protocol for high-performance metadata transfer.

Write. For a write request accepted from applications, module nvme-kglex will pack it into an MP packet and send it to the target server over the GLEX network. The target server accepts it by module nvmet-discovery and processes it by nvmet-core. Module nvmet-core resolves the write packet, transmits it to module blk-mq to get information such as LDA, length, and NQN. Next, target server will fetch data from the memory of the host server according to LDA and length by calling RDMA-get, and write it to its own memory. Finally, target server will write data down to disk chunks according to the information resolved from the MP packet.

I/O request transmission. Once an application issues an I/O request to a host server, the request will be processed by VFS and become a bio (block I/O) command containing the request's meta information such as the LDA (Logical Disk Address) and the length. Then, the bio command will be packed into an nvme command. Finally, both the nvme command and the header (which contains NQN) are packaged into an MP packet, which will be sent to the corresponding

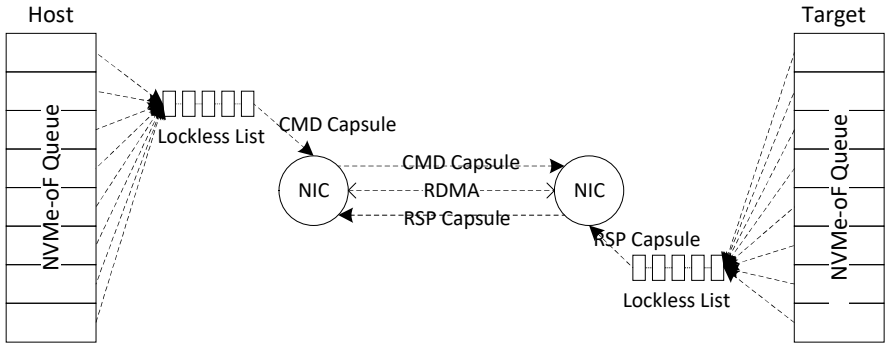


Fig. 8. Structure of nvme-kglex in Host Server. The host server sends NVMe commands (in CMD Capsule) to the target server, which performs disk I/O and sends responses (in RSP Capsule) back to the host server.

target server. The target server resolves the packet. The I/O request will be transmitted to module blk-mq, and the contents of the packet will be resolved to retrieve the bio command, NQN, LDA, and length.

Read. For a read request accepted from applications, the host server will send the request to the target server, which will resolve the packet (similar to the processing of write requests) and transmit the request to blk-mq. The difference is that for write requests the target server fetches data from the memory of host server to its own memory by calling RDMA-get, while for read requests the target server first copies the data read from its disk to its memory, and then copies the data from its memory to the memory of host server by calling RDMA-write.

D. Interrupt Aggregation

In traditional cloud block storage systems (including URSA, Ceph, Blizzard, etc.), when the backend SSD of a target server completes an I/O request, its NVMe controller will send a message signaled interrupt (MSI) that directly writes the interrupt vector of the target server's CPU core, which subsequently executes the ISR (interrupt service routine) associated with the vector's interrupt request (IRQ). Although having worked well for cloud block storage systems, NVMe's interrupt-based notification mechanism cannot adapt to the high-performance I/O requirement on supercomputers because interrupts consume too many CPU cycles to check the I/O completion events [35]. Variations like polling (supported by Linux kernel) and selective interrupt service routine suffer from even more severe CPU usage problems [41] and thus may not be feasible for target servers serving multiple host servers. Even worse, the block I/O processing at the kernel's storage layer, such as requests reordering and merging, also consumes a large number of CPU cycles.

To alleviate the high CPU usage problem of target servers when processing I/O-intensive workloads, UrsaX adopts the `None` scheduler of the kernel to reduce CPU occupation. This has two potential issues, namely, the number of accesses to the device will slightly increases (which is not a problem for modern NVMe SSDs), and the number of interrupts for notification of I/O completions will also increases. To address the second problem (and even better, to reduce the number of interrupts), we leverage the interrupt coalescing feature of the NVMe protocol to ask NVMe controller to *adaptively* lower the interrupt rate, as introduced below.

The NVMe protocol provides two configurable parameters, namely, the aggregation threshold and the aggregation time, for controlling interrupt coalescing. The target server can use the aggregation threshold to tell the NVMe controller about the (expected) minimum number of interrupts for coalescing, and use the time threshold to tell the NVMe controller about the (expected) timeout for issuing a coalesced interrupt. The influence of interrupt coalescing on block I/O performance is two-fold. On the upside, interrupt coalescing effectively reduces the number of interrupts which reduces CPU usage and thus improves I/O throughput; while on the downside, it might also increase I/O latency especially when the the workloads are less intensive.

UrsaX adopts interrupt aggregation for reducing target server CPU usage without severely affecting I/O latency. We observe that for I/O-intensive workloads the coalescing-caused delay usually is insignificant compared to the real disk I/O delay (which dominates the overall latency). Therefore, UrsaX can aggregate more aggressively when the load is heavy, and vice versa.

We implement interrupt aggregation as follows. We adjust the interrupt number threshold and the timeout for the next period. We test all the combinations of the thresholds by using the `fio` benchmark, and obtain the appropriate values that result in the optimal throughput/latency performance measured on the target servers. The adaptive interrupt coalescing mechanism conforms to the NVMe protocol, and thus is practical to be realized on real NVMe devices. In contrast, existing solutions [33], [34], [41] break the NVMe specification thus being impractical for real usage. For example, Calibrated Interrupts (CI) [34] tries to reduce the impact of interrupt coalescing on latency-sensitive requests. CI lets the software explicitly specify to the device whether a submitted request is latency-sensitive, and the device then "calibrates" interrupts to that request's completion. This breaks the NVMe specification thus requiring complex modification to firmware. Therefore, CI is impractical and it has to emulate calibration without implementation on device.

IV. Implementation

A. Management and Maintenance

UrsaX has two types of metadata for management and maintenance of its block storage service. First, the persistent metadata mainly includes the volume ID, status, size, creation times, etc., which need to be permanently stored for durability. Second, the dynamic metadata mainly includes the information of opened volumes, statistics, positions of data chunks (discussed in Section IV-B), etc., which can tolerate crashes by letting host/target servers report their dynamic metadata in heartbeats. Based on the CRUSH mapping algorithm [40], UrsaX stores persistent metadata in a RocksDB [12] cluster and dynamic metadata in a Memcached [20] cluster, respectively, so as to achieve high metadata insertion, deletion, and query performance.

Unlike metadata management where UrsaX adopts CRUSH for metadata placement, for block data storage UrsaX lets the manager explicitly designate the physical storage devices for a client's VLD. This allows UrsaX to control the mapping between the VLD's data and its actual positions, thus easily achieving I/O load balancing and avoiding the data migration problem suffered by Ceph (Section II-A). Further, UrsaX manager performs topology-aware mapping that allows the client to use its nearby nodes' SSDs as the backend of its VLD. For example, the Tianhe exascale supercomputer has a five-level network [21] that connects its hundreds of thousands of nodes, and the storage nodes are equipped at the rack level. When selecting the target servers of given host servers. the UrsaX manager gives higher priority of storage nodes that are nearer to the host servers, thus minimizing the network's negative impact.

UrsaX runs a group of managers for management and maintenance tasks. First, the *storage service manager* is responsible for maintaining and querying the persistent/dynamic metadata for storage services such as virtual disk creation and expansion. UrsaX runs multiple independent storage service managers for high scalability. Second, the *monitor manager* is for monitoring the status of the block storage service, detecting failed storage servers and I/O requests (for further processing), and recovering various failures. UrsaX also deploys multiple monitor mangers each of which watches a subset of the UrsaX system. Third, UrsaX runs the *cluster master manager* in an HA (high-availability) mode, where only one active master at any time and several stand-by masters may wait in the background.

B. Replication

UrsaX organizes data into chunks, and adopts the primary-backup replication mechanism where each data chunk has one primary replica and multiple backup replicas. The target server storing the primary replica is referred to as the chunk's primary server, and the target servers storing the backup replicas are referred to as the chunk's backup servers. UrsaX differentiates the processing of normal requests and large write requests (the sizes of which are larger than a pre-defined threshold ζ). For normal-sized write requests ($\leq \zeta$), the host servers directly write all the primary/backup replicas to the primary/backup servers. In contrast, for large-sized write requests ($> \zeta$), to avoid the bandwidth of the host servers to become a bottleneck, the host servers write the primary replicas only to the primary servers which will be responsible both for persisting the data locally and for replicating the writes to the backup servers, which persist the data and then acknowledge to the primary servers. By default UrsaX configures ζ = 64KB considering the high bandwidth of the GLEX network.

To improve availability when some replica crashes or is temporarily unavailable, UrsaX supports the HA (high-availability) replication mode where a write can be committed (i.e., the host server returns to the tenant) when a majority (instead of all) of the replicas are successfully persisted, as discussed in Section IV-D. When the HA mode is enabled, for consistency read requests are always processed by the primary replica and when the primary is unavailable a backup replica will (temporarily) become primary. In contrast, when the HA mode is disabled read requests can be processed by any of the primary/backup replicas.

C. Parallelism

The host server divides a volume (VLD) into multiple stripes distributed among target storage servers. Similar to the traditional EC (erasure coding) technique, multiple stripes are organized into one group, so as to accelerate block reads and writes by parallelizing I/O. The mapping from volume to stripes is performed by the managers which ensure multiple stripes not in the same storage server. The processing of I/O requests by UrsaX is non-blocking, *i.e.*, one I/O request will not block the processing of other I/O requests. Like physical block devices, I/O requests are handled allowing out-of-order completion without ordering constraint.

UrsaX issues and handles I/O requests using a pipeline for each sender-receiver pair. Therefore, network delay can be efficiently masked by the requests flying over the wire and waiting in the pipeline buffer. UrsaX reduces the negative impact of inter-node communication as much as possible and network delay will not degrade the overall throughput and IOPS performance.

D. Consistency

In UrsaX, host servers provide remote volumes (each backed by multiple target servers) to client tenants. Normally, a volume is mounted by a single client tenant. UrsaX also allows one volume to be mounted by multiple tenants where the volume's host server services all the tenants and relies on upper-layer file systems (such as Oracle Cluster File System [11] and Lustre File System [17]) to consistently and efficiently handle concurrent writes from different tenants to the same volume.

Each host server holds a lease (with a pre-defined timeout at the scale of seconds) for the volume. After the lease is expired, the host server can renew the lease, otherwise other host servers can get a new lease for the volume. In normal cases, the host server commits the write after receiving acknowledgements of (i) primary replica (from the primary server) and (ii) backup replicas (either from backup servers

for normal-sized writes or from primary server for large-sized writes, as discussed in Section IV-B).

To improve availability when some replica crashes or is temporarily unavailable, UrsaX also supports the HA (high-availability) mode, where the write will also be committed once receiving acknowledgements from a majority of the replicas. For example, for three-way replication with one primary and two backup replicas, the write is committed once one primary replica and at least one backup replica are acknowledged. The cluster manager will be notified with the two-replica write, and will then fix the inconsistency problem (if the temporarily-unavailable replica recovers timely) or allocate a new replica for the write after timeout.

V. Evaluation

A. Overview

In this section, we evaluate UrsaX using two testbeds. The first small testbed has six machines for most micro-benchmarks and real-world-use experiments, and the second large testbed has 44 machines (from our Tianhe exascale supercomputer) for scalability tests. We compare UrsaX to the state-of-the-art URSA [26] and Ceph [40] block storage systems, respectively in latency, IOPS, and throughput. Each machine has eight 8-core Phytium FT2000+ 2.30GHz CPU, 128GB RAM, four DERA D5457 3.2TB PCIe SSDs, connected to the 100Gbps GLEX network where UrsaX runs RDMA and URSA and Ceph run TCP.

Note that although RDMA has been supported by a variation of Ceph [36], in our experiments we use the stable official version [1] instead of the variation, since (i) the variation is unavailable online and (ii) the I/O performance of the RDMA-based variation is only $12\% \sim 17\%$ higher than that of the official TCP-based Ceph. The marginal performance gain is mainly because the variation's integration of RDMA is suboptimal [36]: the variation adopts two polling threads (one Ceph epoll-based asynchronous driver thread and one RDMA polling thread) which interfere with each other; and it has to perform one copy from RDMA `recv` buffer to Ceph asynchronous driver buffer.

The block I/O performance is measured by using `fio` running in client tenants which mount the volumes (VLDs) provided by UrsaX, URSA, and Ceph, respectively. We adopt three-way replication (one primary and two backup) and enable the HA mode (Section IV-B). UrsaX stores all the replicas on SSDs. URSA and Ceph run in their SSD-only mode where SSDs are used for both primary and backup replicas, which is the same as UrsaX. For UrsaX, one machine co-runs the storage service manager, monitor manager, and cluster master manager (Section IV-A). For URSA, journals between primary and backup replicas are disabled because they are designed only for bridging the SSD-HDD performance gap in the SSD-HDD-hybrid mode. If not specified, adaptive interrupt aggregation (§III-D) is enabled. Each result is the mean of 10 runs. The OS is RedHat 7.8 (Linux kernel 4.19.46).

In this section, we seek to answer these following questions: Compared to URSA and Ceph how does UrsaX perform in micro benchmarks for I/O performance of latency, IOPS, and throughput (Section V-B)? What is the impact of the GLEX network on UrsaX's I/O performance (Section II-C)? Is UrsaX scalable as the system scale increases (Section V-D)? And how does UrsaX-backed VLDs perform in real-world use compared to local disks (Section V-E?)

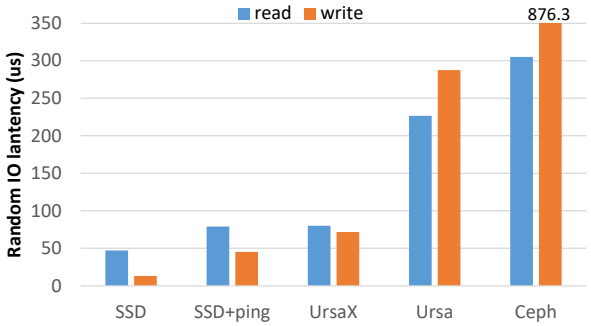


Fig. 9. UrsaX vs. URSA and Ceph in random I/O latency (QD = 1, BS = 4KB).

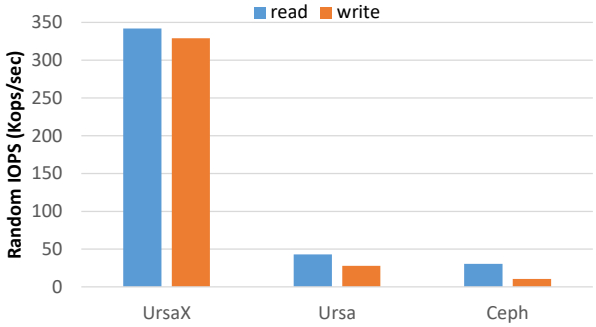


Fig. 10. UrsaX vs. URSA and Ceph in random IOPS (QD = 16, BS = 4KB).

B. Micro Benchmarks

UrsaX efficiently integrates on-disk block I/O and in-network message transfer on supercomputers. To verify UrsaX's performance advantage, we use three machines to run target servers and three machines to run host servers. We evaluate the mean I/O latency of UrsaX running three clients co-located with the three host servers, and compare it to that of URSA and Ceph. In this test, the qd (queue depth) is 1, and the block size is 4KB ($< \zeta = 64$KB discussed

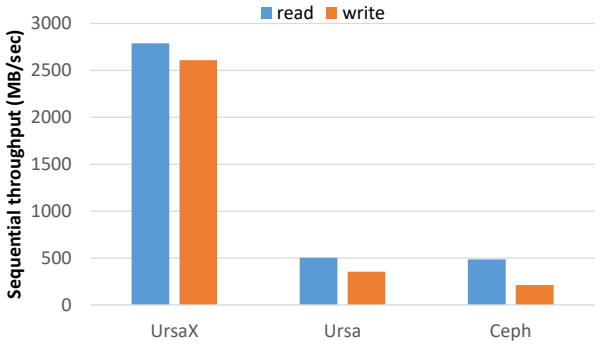


Fig. 11. UrsaX vs. URSA and Ceph in sequential throughput (QD = 1, BS = 1MB).

in Section IV-B) making the host servers directly replicate primary/backup replicas to primary/backup servers in parallel.

The latency result is shown in Fig. 9, where we also include the latencies of local SSD and SSD plus network ping for comparison. UrsaX's latency of both read and write is less than 100 microseconds, which is very close to the optimal latency (SSD plus ping) and is several times lower than that of URSA (higher than 200 microseconds). This is mainly because UrsaX effectively leverages the hardware features (like NVMe-Fabrics and GLEX network) of the supercomputer, while URSA is completely agnostic to the underlying supercomputer and only performs cloud-style network communication and disk I/O. The latency of Ceph is much higher than that of UrsaX and URSA, because Ceph's block storage is built on top of its object abstraction layer which precludes opportunities for network and disk I/O optimization.

We then evaluate the IOPS of UrsaX using three host servers and three target servers, and compare it with that of URSA and Ceph. the qd (queue depth) is 16 for pipelining. The block I/O size is 4KB in the IOPS test which is smaller than $\zeta = 64$KB, and thus the host servers directly write replicas to primary/backup servers in parallel. The mean IOPS is shown in Fig. 10, which shows that UrsaX achieves significantly higher IOPS than URSA and Ceph. It confirms that the integrated design of storage and network layers greatly improves IOPS. Besides, the CPU usage of UrsaX is 13.2% and 16.1% lower than that of UrsaX without interrupt aggretation (not shown in the figure), demonstrating UrsaX reduces the involvement of target servers' CPUs in block I/O service.

We also evaluate the throughput of UrsaX using three host servers and three target servers, and compare it with that of URSA and Ceph. the qd (queue depth) is 1, and the block I/O size is 1MB in the throughput test which is larger than $\zeta =$ 64KB making the host servers only write primary replicas to primary servers and rely on the primary servers to replicate backup replicas to backup servers. The mean throughput is shown in Fig. 11. UrsaX achieves more than $5\times$ higher throughput compared to URSA and Ceph, because UrsaX is more efficient than URSA and ceph in exploiting the high bandwidth of supercomputers' storage and network.

The latency, IOPS and throughput of URSA and Ceph on the supercomputer nodes are similar with that on commodity machines (reported in Ref. [26]), mainly because their storage/network software stacks introduce high overhead. In contrast, UrsaX fully leverages the supercomputer's hardware advantage thus being more suitable for block storage service on supercomputers.

C. Network Impact

The core of UrsaX is the integration of its storage and network designs, where UrsaX utilizes NVMe-Fabrics on the GLEX network with separate metadata and data I/O. In this section we evaluate the effectiveness of the integration by evaluating the impact of the GLEX network on UrsaX's I/O performance. To measure UrsaX's I/O performance *without* network communication, we co-run host servers and target servers on all the six machines of the small testbed, and statically map a host server's VLDs onto the physical disks of a target server which is co-located with the host server on the same machine. We measure the mean local I/O performance and compare it with UrsaX's normal I/O performance where the host/target interaction is across the GLEX network.

Fig. 12. Impact of GLEX network on UrsaX IOPS for various block sizes.

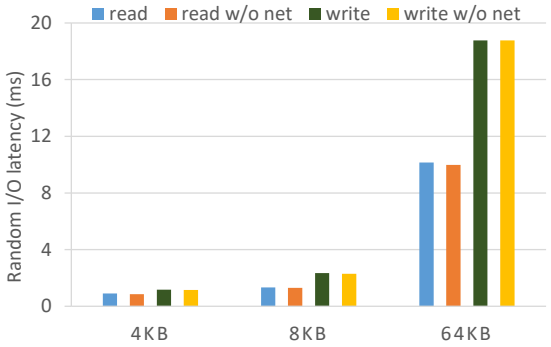


Fig. 13. Impact of GLEX network on UrsaX latency for various block sizes.

We run the test to evaluate the IOPS for various queue depths (qd) of $1, 2, 4, \cdots, 256$, and find that normal I/O and local I/O of UrsaX always have similar performance. Fig. 12 shows the mean IOPS with qd=64 for different block sizes of 4KB, 8KB, and 64KB. The results show that the IOPS of normal reads/writes of UrsaX is only slightly lower than that of its local reads/writes, which proves that the extra cost of cross-network I/O is small and demonstrates the effectiveness of UrsaX's storage-network-integrated design.

We also measure the I/O latency when evaluating IOPS in the above experiment. The result is shown in Fig. 13, where the normal read/write latency is very close to that of the local read/write latency, demonstrating that the extra latency overhead induced by the GLEX network is low owing to UrsaX's integrated design of the storage/network layers. The result shows that UrsaX achieves high IOPS without compromising the latency performance, to which the traditional HPC applications are very sensitive.

D. Scalability

The design of host/target servers allows UrsaX to be horizontally scalable. To verify the scalability of UrsaX, we first measure the aggregate IOPS of UrsaX as the number of storage machines increases from 11 to 44. Clients, host servers and

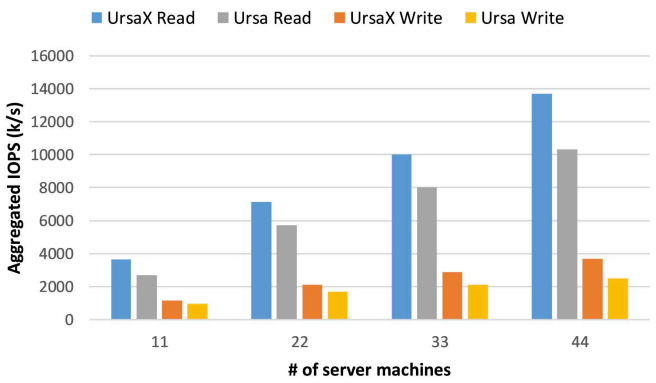


Fig. 14. Random IOPS of UrsaX and URSA as the system scale increases from 11 to 44 (QD = 16, BS = 4KB).

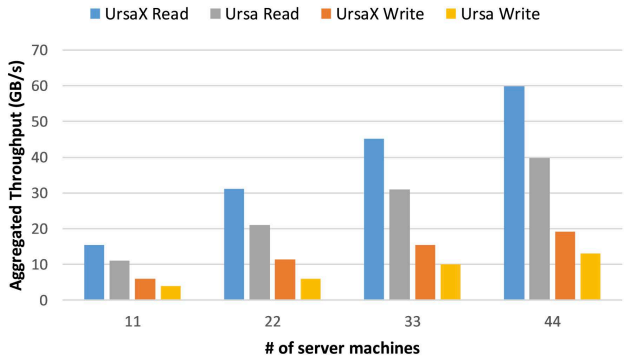


Fig. 15. Sequential throughput of UrsaX and URSA as the system scale increases from 11 to 44 (QD = 1, BS = 1MB).

target servers co-locate on all machines to saturate the entire system. The qd is 16 and the block size is 4KB for the IOPS test.

The result is shown in Fig. 14, where we also include the IOPS of URSA for comparison. The IOPS of UrsaX and URSA both increase linearly with the number of machines, but the IOPS of UrsaX is always significantly higher than URSA because UrsaX can fully exploit the potential of the supercomputer.

We also evaluate the aggregate throughput of UrsaX and URSA as the number of storage machines increases from 11 to 44. The qd is 1 and the block size is 1MB for the throughput test. The results are shown in Fig. 15. Similar to the IOPS test, the throughput of UrsaX increases linearly with the number of machines and is always much higher than that of URSA.

E. Applications

UrsaX has been deployed on a subset (containing hudreds of nodes) of our supercomputer to provide block storage service for many tenants running not only traditional HPC applications but also supercomputer-oblivious POSIX applications. In this subsection we evaluate two real-world applications (a cluster file system and a key-value store) that use the VLDs provided by UrsaX as their block storage devices.

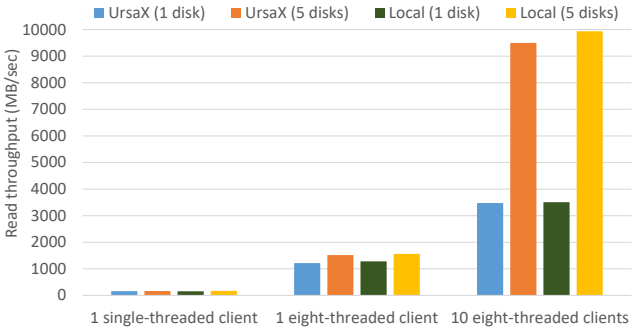


Fig. 16. UrsaX vs. local disks for read throughput in Lustre filesystem using IRO (file size = 1MB).

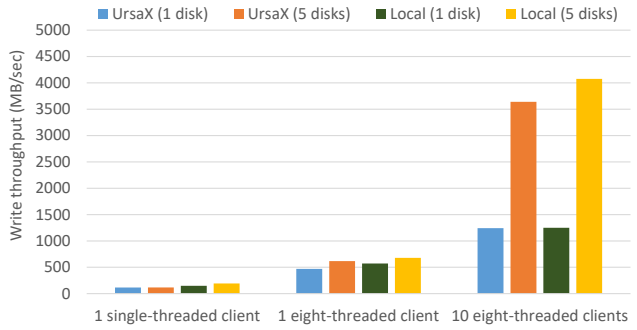


Fig. 17. UrsaX vs. local disks for write throughput in Lustre filesystem using IRO (file size = 1MB).

1) Lustre File System: Lustre [17] is a parallel cluster file system widely used on supercomputers including Fugaku [16], Titan [18], Sequoia [15]. We run the Luster file system (v2.12.5) with five VLDs each from one host server. We also run the Luster file system with five physical disks each on one target server. For fairness in this experiment each VLD from a host server is statically mapped onto three physical disks each on one target server (for three-way replication). The version of the backend ZFS of Lustre is v0.8.6-1. We use the IRO [9] benchmark tool to respectively run one single-threaded client, one eight-threaded client, and 10 eight-threaded clients, performing 1MB-file-size read/write requests in parallel (totally 1TB). The multi-client tests have concurrent I/O requests to the Luster file system.

Figs. 16 and 17 respectively compare the read and write throughput of UrsaX-backed and local-disk-backed Lustre file system, where we also include the throughput for one VLD and local disk for comparison. The results show that UrsaX-backed Lustre file system have almost the same throughput as the local-disk-backed one, which demonstrates the effectiveness of UrsaX's storage-network-integrated design even for concurrent I/O workloads supported by the widely-used Luster file system.

2) RocksDB: RocksDB [12] is a high-performance key-value (KV) store. It leverages a log-structured merge-tree (LSMT) structure to make efficient use of SSDs for I/O-intensive workloads. RocksDB writes KVs both in an in-memory Memtable (of size = `write_buffer_size`) and *asynchronously* to an on-disk write ahead log (WAL). When

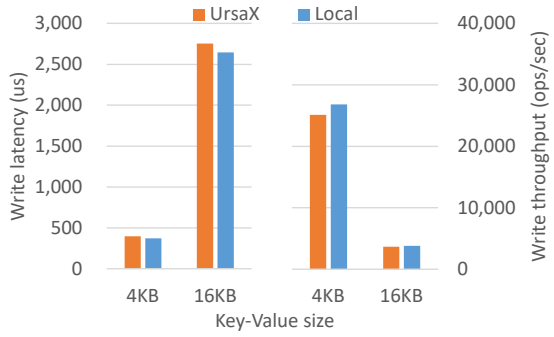


Fig. 18. UrsaX vs. local disks for KV write performance in RocksDB.

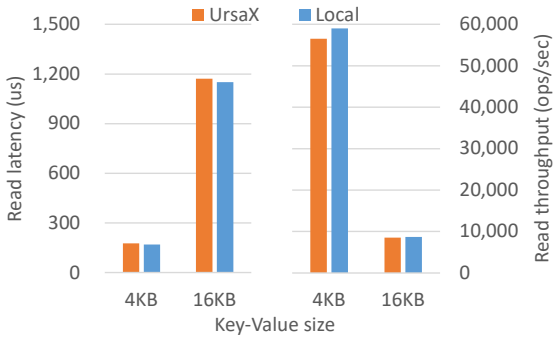
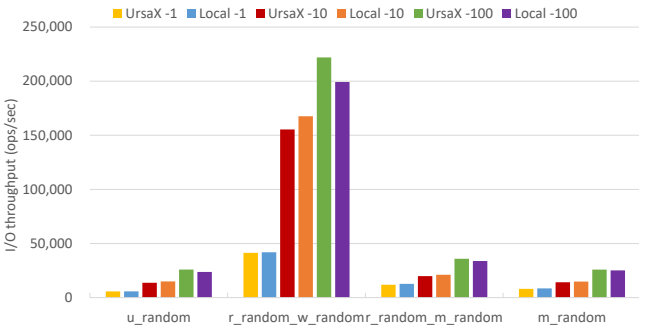
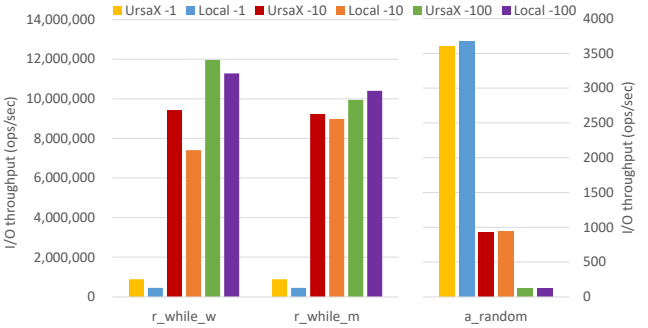


Fig. 19. UrsaX vs. local disks for KV read performance in RocksDB.



(a) Workloads: update_random, read_random_write_random, read_random_merge_random, merge_random



(b) Workloads: read_while_write, read_while_merge, append_random

Fig. 20. UrsaX vs. local disks for various workloads in RocksDB using `db_bench` (KV size = 4KB).

a Memtable is full, it is flushed to a Sorted Sequence Table (SST) file and the log is cleaned. SST files are organized in a sequence of levels starting from Level-0.

We compare the I/O performance of RocksDB respectively running a VLD provided by UrsaX and a physical disk (both with the `ext4` file system). To eliminate the impact of memory-cached KV writes, we disable the in-memory Memtable (`write_buffer_size` = 0 and `sync` = `true`). Similar to the Luster file system test (Section V-E1), in this experiment we also map each VLD from a host server onto three physical disks each on one target server.

We measure the mean I/O latency and throughput of RocksDB respectively on an UrsaX-backed VLD and a physical disk, by putting/getting one million KV with sizes of 4KB and 16KB. The comparison result (between UrsaX and the physical disk) for write is shown in Fig. 18, where UrsaX achieves almost the same latency and throughput of RocksDB as the physical disk. The comparison result for read is similar to that for write, thus being omitted due to lack of space.

We also use the `db_bench` KV benchmark tool to evaluate more realistic KV I/O workloads of RocksDB with the number of threads $N = 1, 10, 100$. The size of key-value pairs is 4KB. The workloads include `update_random` (N threads doing read-modify-write for random keys), `read_random_write_random` (N threads doing random read and write), `read_random_merge_random` (perform N random read-or-merge operations), `merge_random` (same as `update_random` but using merge operator), `read_while_write` (1 writer, N threads doing random reads), `read_while_merge` (1 merger, N threads doing random reads), and `append_random` (N threads doing read-modify-write with growing values). The results are shown in Fig. 20, where the read/write throughput of RocksDB on UrsaX is always close to that on the physical disks. UrsaX outperforms local disk for some compound read-intensive workloads (`read_while_write` and `read_while_merge`), mainly because UrsaX adopts three-way replication and each replica can serve the read request.

VI. Conclusion

This paper presents UrsaX, an efficient block storage service on GLEX network for our next-generation Tianhe exascale supercomputer. At the core of UrsaX is the efficient integration of on-disk block I/O and in-network message transfer on supercomputers. UrsaX utilizes the NVMe-Fabrics kernel module to expand the NVMe standard on the GLEX network, and separates metadata I/O and data I/O of blocks respectively being handled over the MP and RDMA protocols.

In our future work, we will apply UrsaX on the entire exascale supercomputer and conduct extensive evaluation on hundreds of thousands of nodes. We also plan to extend the design of storage-network-integration of UrsaX to a general remote block service which can be migrated to other mainstream supercomputers with minimal effort.

Acknowledgements

This work is supported by the National Key R&D Program of China (2022YFB4500302), the National Natural Science Foundation of China (61932001), and the Foundation of State Key Lab of HPC (202101-03). Zhenlong Song is the corresponding author.

REFERENCES

[1] "http://ceph.com/."
[2] "https://aws.amazon.com/ebs/."
[3] "https://azure.microsoft.com/en-us/services/storage/."
[4] "https://docs.ceph.com/en/latest/rbd/."
[5] "https://docs.ceph.com/en/quincy/rados/index.html."
[6] "https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html."
[7] "https://iadac.github.io/events/adac8/sparks.pdf."
[8] "https://intl.aliyun.com/."
[9] "https://ior.readthedocs.io/en/latest/."
[10] "https://kubernetes.io/."
[11] "https://oss.oracle.com/projects/ocfs/."
[12] "https://rocksdb.org/."
[13] "https://sheepdog.github.io/sheepdog/."
[14] "https://wiki.openstack.org/cinder."
[15] "https://www.energy.gov/articles/sequoia-ranked-fastest-supercomputer-world."
[16] "https://www.fujitsu.com/global/about/innovation/fugaku/."
[17] "https://www.lustre.org/."
[18] "https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/."
[19] "https://www.qcloud.com/."
[20] "http://www.memcached.org/."
[21] X. Gan, Y. Zhang, R. Zeng, J. Liu, R. Wang, T. Li, L. Chen, and K. Lu, "Xtree: Traversal-based partitioning for extreme-scale graph processing on supercomputers," in *IEEE ICDE*, 2022.
[22] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *FAST 14*, 2014, pp. 199–212.
[23] D. Hildebrand and P. Honeyman, "Exporting storage systems in a scalable manner with pnfs," in *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*. IEEE, 2005, pp. 18–27.
[24] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–15.
[25] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 84–92.
[26] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong, "Ursa: Hybrid block storage for cloud-scale virtual disks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.
[27] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Parallax: virtual disks for virtual machines," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 41–54.
[28] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy, "Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 257–273.
[29] P. Mishra, T. Agrawal, and P. Malakar, "Communication-aware job scheduling using slurm," *49th International Conference on Parallel Processing - ICPP : Workshops*, 2020.
[30] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, , and Y. Suzue, "Flat datacenter storage," in *OSDI*, 2012.
[31] S. Peng, X. Zhang, W. Su, D. Dong, Y. Lu, X. Liao, K. Lu, C. Yang, J. Liu, W. Zhu, and D. Wei, "High-scalable collaborated parallel framework for large-scale molecular dynamic simulation on tianhe-2 supercomputer," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 17, pp. 804–816, 2020.
[32] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 497–514.
[33] A. Tai, I. Smolyar, M. Wei, and D. Tsafrir, "Barrier-enabled IO stack for flash storage," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 129–145. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/tai
[34] A. Tai, I. Smolyar, M. Wei, and D. Tsafrir, "Optimizing storage performance with calibrated interrupts," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 129–145. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/tai
[35] A. Tai, I. Smolyar, M. Wei, and D. Tsafrir, "Optimizing storage performance with calibrated interrupts," *ACM Transactions on Storage (TOS)*, vol. 18, no. 1, pp. 1–32, 2022.
[36] H. Tang, J. Zhang, and F. Zhang, "Accelerating ceph with RDMA and nvme-of," in *14th OpenFabrics Alliance Annual Workshop*. OpenFabrics, 2018.
[37] L. Wang, Y. Zhang, J. Xu, and G. Xue, "Mapx: Controlled data migration in the expansion of decentralized object-based storage systems," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 1–11.
[38] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the salus scalable block store," in *NSDI*, 2013, pp. 357–370.
[39] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: Managing storage for a million machines." in *HotOS*, 2005.
[40] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006, pp. 307–320.
[41] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung, "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in *OSDI*, Carlsbad, CA, 2018, pp. 477–492.